Operating Systems

Lecture 2
Interrupt & Syscall

IIIS & CS
Tsinghua University

Acknowledgement:
materials from Dr. Zhang Yong Guang in MSRA,
And from http://williamstallings.com/OS/OS5e.html , http://www.os-book.com

Outline

- Interrupts & Exceptions
  - Background
  - Interrupt v.s. Exception
  - Interrupt Process Mechanism
  - Nested Execution
- Syscall
  - Concept of System Calls
  - System Call Implementation
  - Difference between routine call and system call
- System boot

Background

- kernel is **trusted** third-party that runs the machine
- Only the kernel can execute **privileged** instructions

System Call

- How can a user program change to the kernel address space?
- How can the kernel transfer to a user address space?

Device Interrupt

- What happens when a device attached to the computer needs attention?

Background

- System Call (aka: Trap from user program)
  - a system call invoked by a user program
- Exception (from ill program)
  - an illegal instruction or other kind of bad processor state (memory fault, etc.).
- Interrupt (from device)
  - timer/net interrupt from different hardware device.

Interrupts vs Exceptions

- Hardware support for getting CPUs attention
  - Often transfers from user/kernel to kernel mode

    Nested interrupts are possible; interrupt can occur while an interrupt handler is already executing (in kernel mode)
  - Asynchronous: device or timer generated hardware event

    Unrelated to currently executing process
  - Synchronous: immediate result of last instruction

    Often represents a hardware error condition
- Intel terminology and hardware
  - Irqs, vectors, IDT, gates, PIC, APIC
- Interrupt handling: data structures, flow of control
- Delayed Handlers: softirqs, tasklets, bottom halves

Interrupts vs Exceptions

- Similar to context switch (but lighter weight)
  - Hardware saves a small amount of context on stack
  - Includes interrupted instruction if restart needed
  - Execution resumes with special "iret" instruction
- Structure: top and bottom halves
  - Top-half: do minimum work and return
  - Bottom-half: deferred processing
- Handler code executed in response
  - Possible to temporarily mask interrupts
  - Handlers need not be reentrant
  - But other interrupts can occur, causing nesting

Interrupt Process Mechanism

- An interrupt is an internal or external event that forces a hardware call to a function called an interrupt service routine.

<u>hardware</u>

- **Interrupt Enable Flag** must be set [CPU initialization]
  1. Internal or external event forces interrupt flag to be set
  2. Event forces routine at interrupt vector to be called

Interrupt Process Mechanism

software

- Processor state must be preserved [compiler]
- Interrupt service routine (ISR) must process data [os developer's code]
- Interrupt flag must be cleared [os developer's code]
- Processor state must be restored [compiler]

Nested Execution

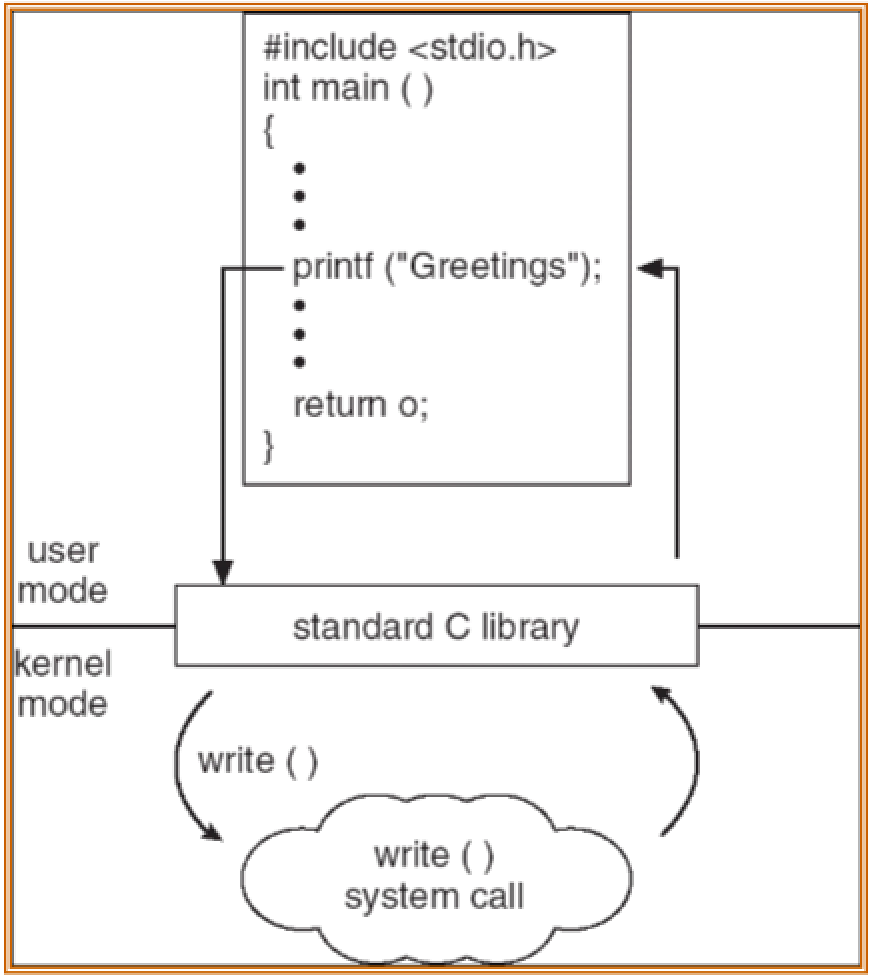
- Interrupts can be interrupted
  - By different interrupts; handlers need not be reentrant
  - No notion of priority in Linux
  - Small portions execute with interrupts disabled
  - Interrupts remain pending until acked by CPU
- Exceptions can be interrupted
  - By interrupts (devices needing service)
- Exceptions can nest two levels deep
  - Exceptions indicate coding error
  - Exception code (kernel code) shouldn't have bugs
  - Page fault is possible (trying to touch user data)

Outline

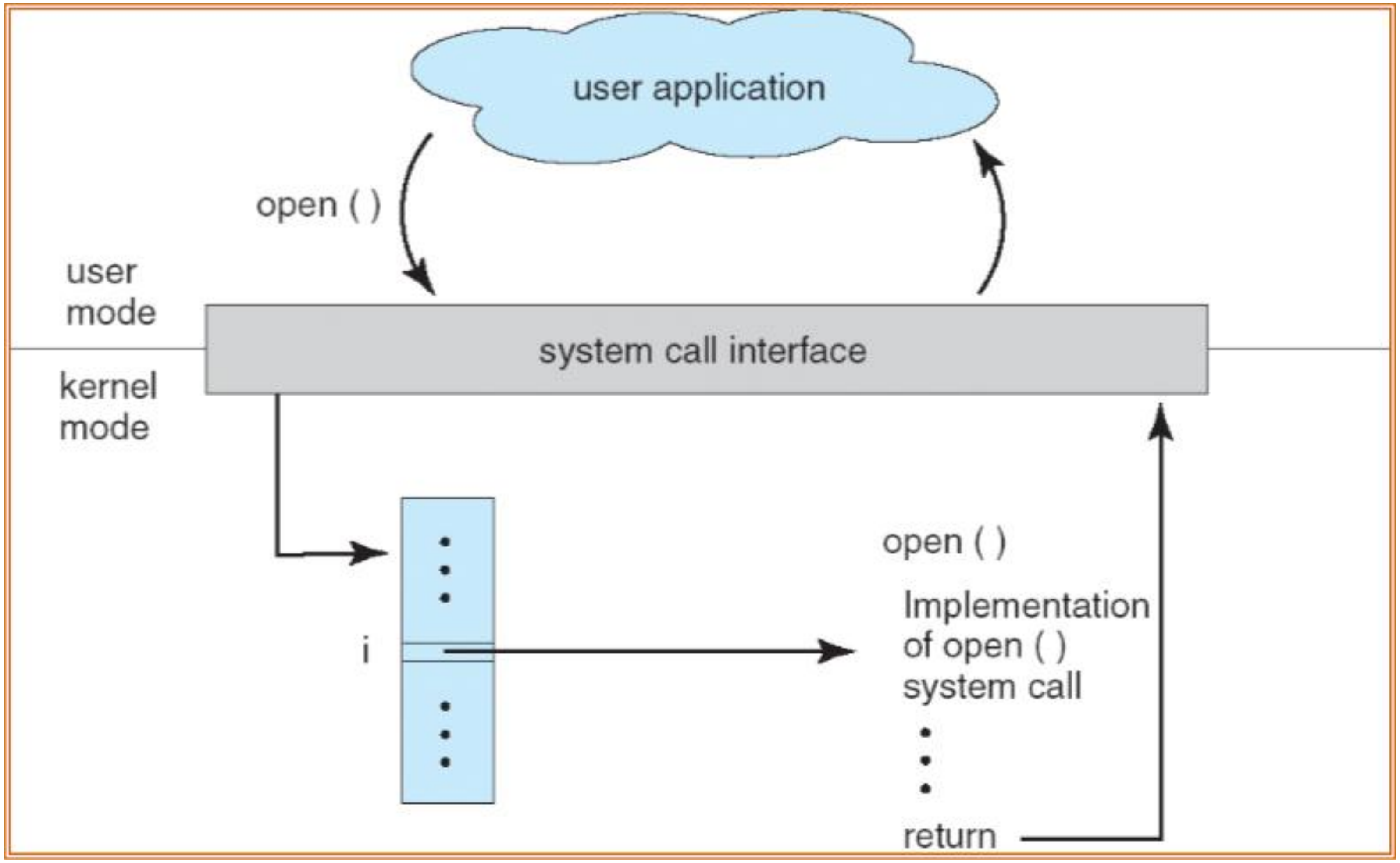
- Interrupts & Exceptions
  - Background
  - Interrupt v.s. Exception
  - Interrupt Process Mechanism
  - Nested Execution
- Syscall
  - Concept of System Calls
  - System Call Implementation
  - Difference between routine call and system call
- System boot

Standard C Library Example

- C program invoking printf() library call, which calls write() system call



API – System Call – OS Relationship

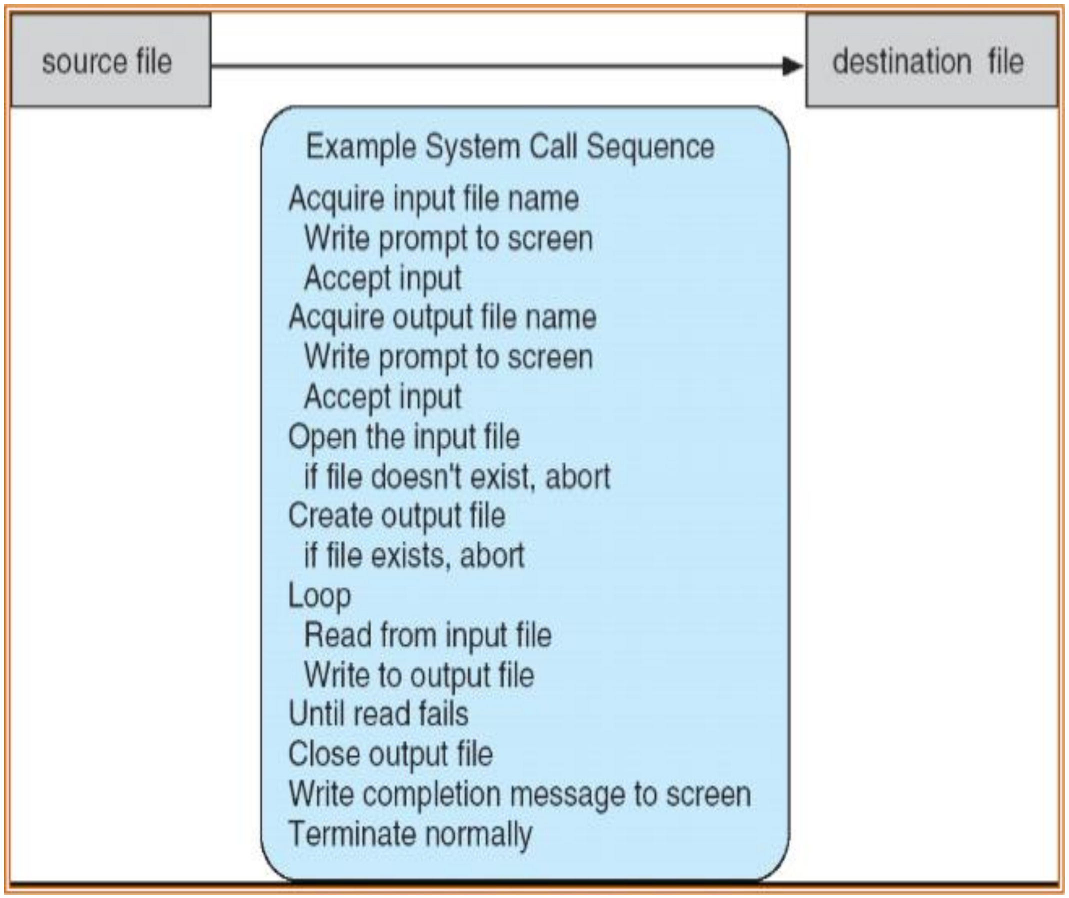


System Calls

- Programming interface to the services provided by the OS
- Typically written in a high-level language (C or C++)
- Mostly accessed by programs via a high-level Application Program Interface (API) rather than direct system call use
- Three most common APIs
  - Win32 API for Windows
  - POSIX API for POSIX-based systems (including virtually all versions of UNIX, Linux, and Mac OS X)
  - Java API for the Java virtual machine (JVM)

Example of System Calls

- System call sequence to copy the contents of one file to another file




```
// System call numbers
#define SYS_fork    1
#define SYS_exit    2
#define SYS_wait    3
#define SYS_pipe    4
#define SYS_write   5
#define SYS_read    6
#define SYS_close   7
#define SYS_kill    8
#define SYS_exec    9
#define SYS_open    10
#define SYS_mknod   11
#define SYS_unlink 12
#define SYS_fstat  13
#define SYS_link    14
#define SYS_mkdir  15
#define SYS_chdir  16
#define SYS_dup     17
#define SYS_getpid 18
#define SYS_sbrk    19
#define SYS_sleep  20
#define SYS_procmem 21
```

Example of user-level syscall API

- Consider read() function in ucore—a function for reading from a file

  in user/libs/file.h: int read(int fd, void * buf, int length)

- A description of the parameters passed to read()

  int fd—the file to be read

  void * buf—a buffer where the data will be read into and written from

  int length—the number of bytes to be read into the buffer

  int return_value: the number of bytes that readed

- Example

  in sfs_filetest1.c: ret = read(fd, data, len);

Example of user-level syscall API

- in sfs_filetest1.c: ret=read(fd,data,len);

```
 ……
 8029a1:   8b 45 10            mov    0x10(%ebp),%eax
 8029a4:   89 44 24 08         mov    %eax,0x8(%esp)
 8029a8:   8b 45 0c            mov    0xc(%ebp),%eax
 8029ab:   89 44 24 04         mov    %eax,0x4(%esp)
 8029af:   8b 45 08            mov    0x8(%ebp),%eax
 8029b2:   89 04 24            mov    %eax,(%esp)
 8029b5:   e8 33 d8 ff ff      call   8001ed <read>
syscall(int num, ...) {
...
        asm volatile (
                         "int %1;"
                         : "=a" (ret)
                         : "i" (T_SYSCALL),
                           "a" (num),
                           "d" (a[0]),
                           "c" (a[1]),
                           "b" (a[2]),
                           "D" (a[3]),
                           "S" (a[4])
                         : "cc", "memory");
        return ret;
```

System Call Implementation

- Typically, a number associated with each system call
  - System-call interface maintains a table indexed according to these numbers
- The system call interface invokes intended system call in OS kernel and returns status of the system call and any return values
- The caller need know nothing about how the system call is implemented
  - Just needs to obey API and understand what OS will do as a result call
  - Most details of OS interface hidden from programmer by API
    - Managed by run-time support library (set of functions built into libraries included with compiler)

Steps in Making a System Call

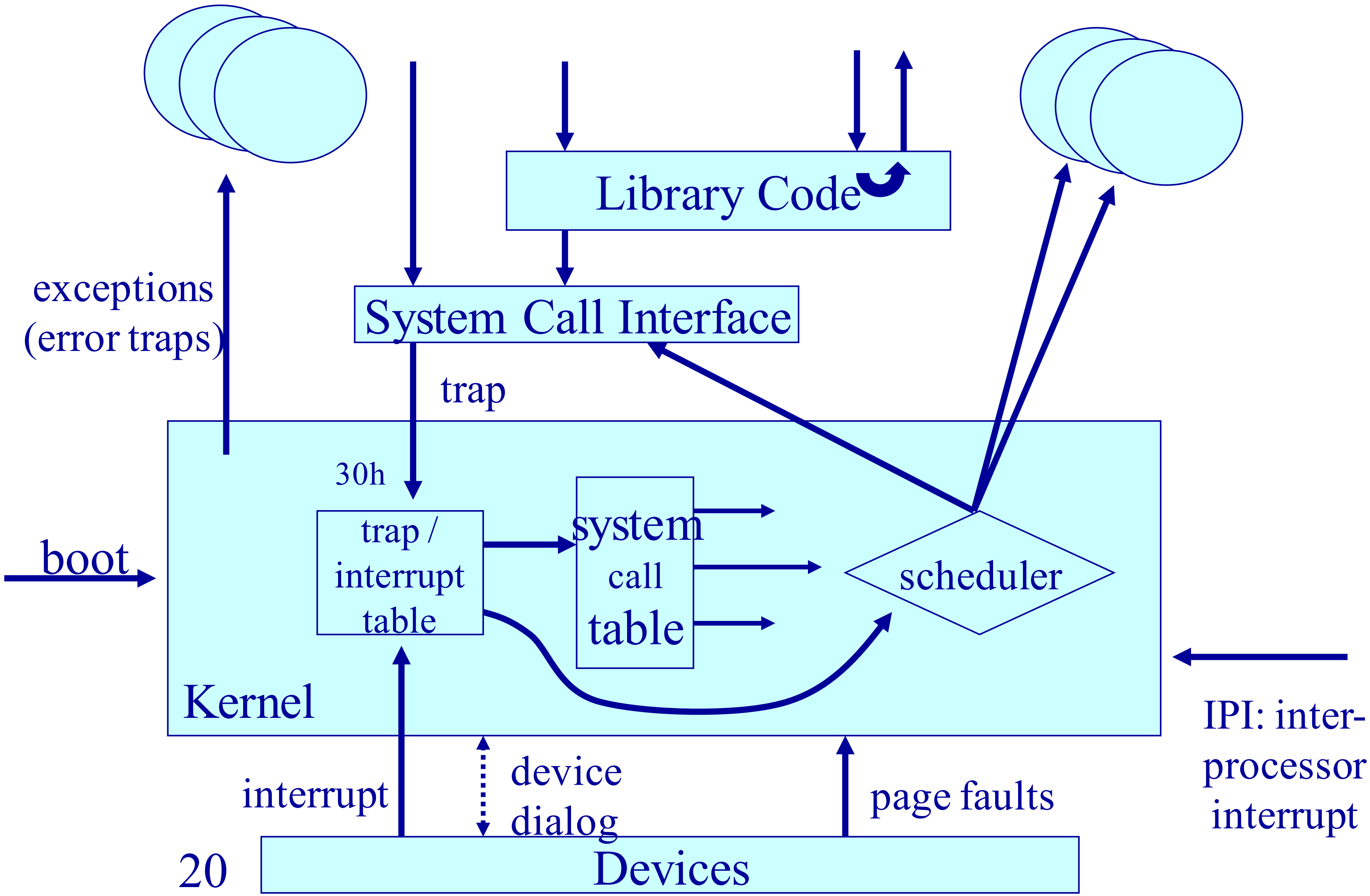
1. alltraps(): in kern/trap/trapentry.S
2. trap(): tf->trapno == T_SYSCALL , in kern/trap/trap.c
3. syscall(): tf->tf_regs.reg_eax ==SYS_read, in /kern/syscall/syscall.c
4. sys_read(): get fd, buf, length from tf->sp, in kern/syscall/syscall.c
5. sysfile_read(): read file content, in kern/fs/sysfile.c
6. trapret(): in kern/trap/trapentry.S

There are 6 important steps in making the system call read (fd, buffer, length)

System Call Parameter Passing

- Often, more information is required than simply identity of desired system call
  - Exact type and amount of information vary according to OS and call
- Three general methods used to pass parameters to the OS
  - Simplest: pass the parameters in *registers*
    - In some cases, may be more parameters than registers
  - Parameters stored in a *block*, or table, in memory, and address of block passed as a parameter in a register
    - This approach taken by Linux and Solaris
  - Parameters placed, or *pushed*, onto the *stack* by the program and *popped* off the stack by the operating system (ucore methord)
  - Block and stack methods do not limit the number or length of parameters being passed

Kernel Entry and Exit



OS Difference between routine call and system call

- int or trap are used for system call
  - Ring level and stack are switched during system call
- call or jmp are used for routine call
  - No stack switch during routine call

Reference: http://www.intel.com/products/processor/manuals/index.htm

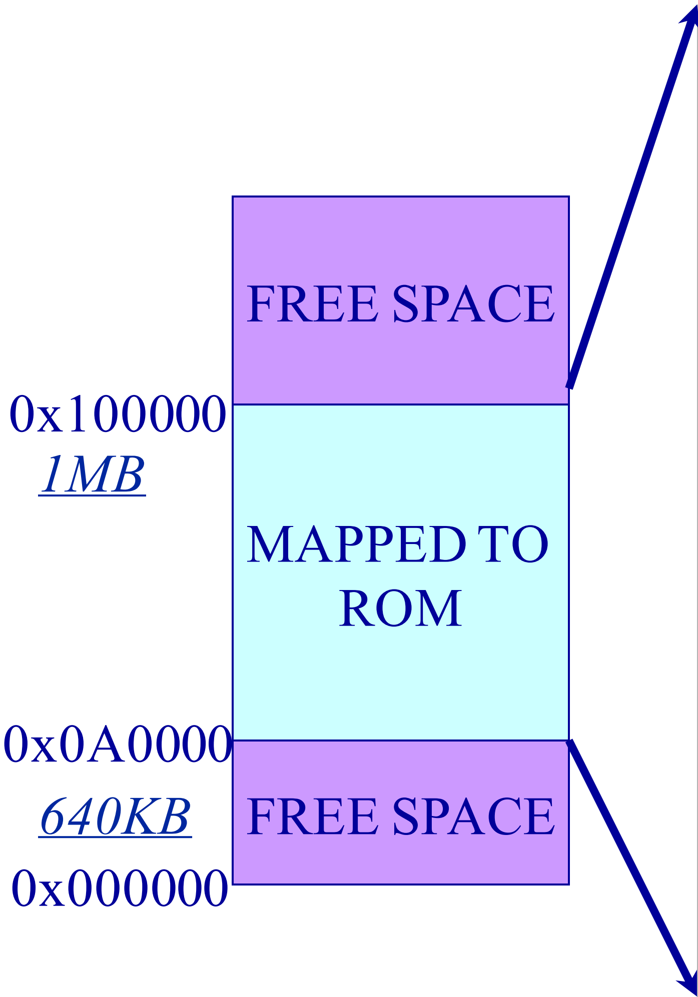
Cost of Crossing the "Kernel Barrier"

- more than a procedure call
- less than a context switch
- costs:
  - vectoring mechanism
  - establishing kernel stack
  - validating parameters
  - kernel mapped to user address space?
    - updating page map permissions
  - kernel in a separate address space?
    - reloading page maps
    - invalidating cache, TLB

Outline

- Interrupts & Exceptions
  - Background
  - Interrupt v.s. Exception
  - Interrupt Process Mechanism
  - Nested Execution
- Syscall
  - Concept of System Calls
  - System Call Implementation
  - Difference between routine call and system call
- System boot

Memory Map at Power up



Address	First 16K (x0000h-x3FFFh)	Second 16K (x4000h-x7FFFh)	Third 16K (x8000h-xBFFFh)	Fourth 16K (xC000h-xFFFFh)
A0000-AFFFFh	VGA Graphics Mode Video RAM			
B0000-BFFFFh	VGA Monochrome Text Mode Video RAM		VGA Color Text Mode Video RAM	
C0000-CFFFFh	VGA Video BIOS ROM		IDE Hard Disk BIOS ROM	Optional Adapter ROM BIOS or RAM UMBs
D0000-DFFFFh	Optional Adapter ROM BIOS or RAM UMBs			
E0000h-EFFFFh	System BIOS Plug and Play Extended Information			
F0000-FFFFFh	System BIOS ROM			

ROM shadowing

- The address space occupied by BIOS Roms is not available as useful RAM.
- BIOS can make use of this RAM to shadow its ROM.
- Write protects this shadowed RAM, disables ROM.

Start up Sequence

- CS:IP = 0xf000:fff0.
- Starts in real mode.
  - 16 bit mode.
  - IP = 16 bit offset
  - CS = Segment offset in 16 byte units.
  - i.e PC = 16*CS+IP
  - Max: 1MB of address space.
  - A20 pain from the past.

Start up Sequence

- POST(Power on self test)
- Looks for video card and executes its BIOS.
- Looks for other option ROMS e.g IDE disk.
- Does more system inventory e.g COM ports, setting hard disk params.
- Plug and play support.
- Sets up IDT and the interrupt service routines

BIOS contd…

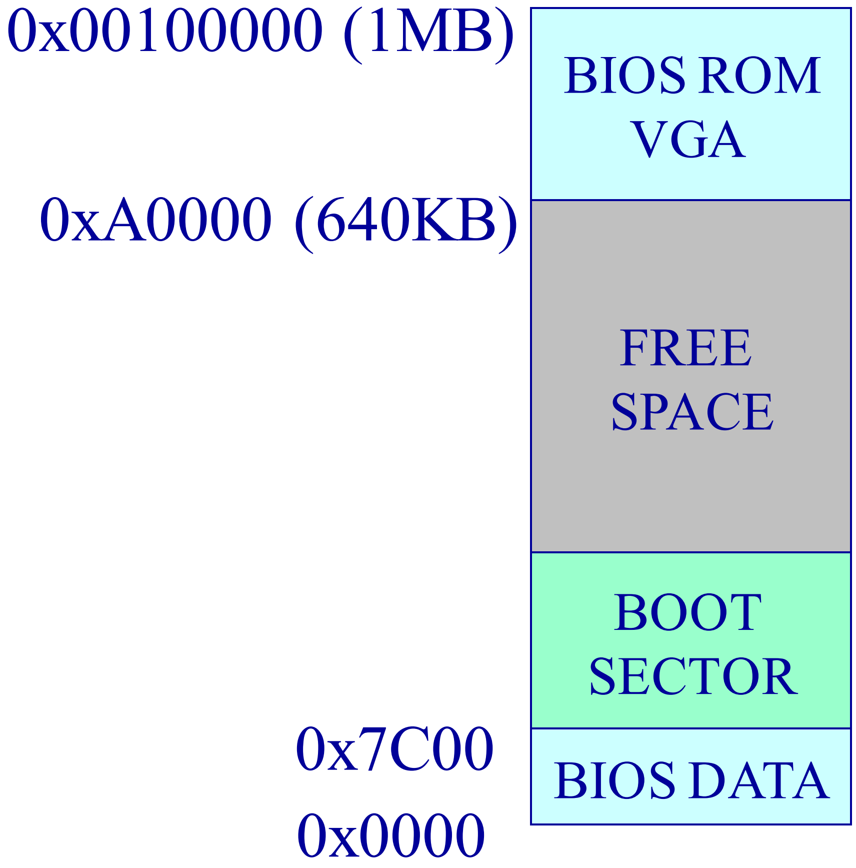
- Looks for bootable media.
- Loads Boot sector(512 bytes) of the media at 0x7c00 .
- Jumps to CS:IP = 0000:7c00 with DL=drive id of bootable drive.

- BIOS data area from 0x0000 to 0x7c00.
(Contains IDT,ISR's and data).

BIOS system calls

- BIOS provides low level I/O routines through interrupts.
- Main services are:
  - INT 15h: Get memory map.
  - INT 13h: Disk I/O interrupts.
  - INT 19h: Bootstrap loader.

Memory Map at this stage



Switches processor into 32-bit mode

```
# Switch from real to protected mode, using a bootstrap GDT
# and segment translation that makes virtual addresses
# identical to physical addresses, so that the
# effective memory map does not change during the switch.
 lgdt    gdtdesc
 movl    %cr0, %eax
 orl     $CR0_PE_ON, %eax
 movl    %eax, %cr0

 # Jump to next instruction, but in 32-bit code segment.
 # Switches processor into 32-bit mode.
 ljmp    $PROT_MODE_CSEG, $protcseg

 .code32                 # Assemble for 32-bit mode
```

This week's Work

- Finish lab #1
- Homework #1